12

Technical Report 1294

Reliable Interconnection Networks for Parallel Computers

Larry R. Dennison

MIT Artificial Intelligence Laboratory

80p8

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
October 1991

3. REPORT TYPE AND DATES COVERED
technical report

4. TITLE AND SUBTITLE
Reliable Interconnection Networks for Parallel Computers

5. FUNDING NUMBERS
N00014-80-C-0622
N00014-85-K-0124
N00014-91-J-1698

6. AUTHOR(S)
Larry R. Dennison

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
AI-TR 1294

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Information Systems
Arlington, Virginia 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution of this document is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A new protocol, the unique token protocol, for reliably transporting data in a network is described. This protocol makes use of existing buffer storage in the network for the replication of data and avoids duplicate elimination at the destination through the use of a token. The unique token protocol is compared to end-to-end protocols in terms bandwidth, latency, and memory requirements, for which it is found to equal or better them. It is also shown to have constant memory requirements per switching and processing element, thus allowing networks employing the protocol to be arbitrarily large. In addition, the organization of a reliable switching element incorporating the protocol is described. A register transfer model of the switching has been implemented. The model and its validation are presented.

14. SUBJECT TERMS (key words)

networks	fault tolerance	parallel computers
reliable	routors	
protocols	virtual channels	

15. NUMBER OF PAGES
78

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UNCLASSIFIED

Reliable Interconnection Networks For Parallel Computers[1]

by
Larry R. Dennison

Submitted to the
Departmentment of Electrical Engineering and Computer Science
on April 18, 1991, in partial fulfillment of
the requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract
A new protocol, the unique token protocol, for reliably transporting data in a network is described. This protocol makes use of existing buffer storage in the network for the replication of data and avoids duplicate elimination at the destination through the use of a token. The unique token protocol is compared to end-to-end protocols in terms bandwidth, latency, and memory requirements, for which it is found to equal or better them. It is also shown to have constant memory requirements per switching and processing element, thus allowing networks employing the protocol to be arbitrarily large. In addition, the organization of a reliable switching element incorporating the protocol is described. A register transfer model of the switching has been implemented. The model and its validation are presented.

Thesis Supervisor: Dr. William J. Dally
Title: Associate Professor of Electrical Engineering and Computer Science
Keywords: Networks, reliable, protocols, routers, virtual channels, parallel computers, fault tolerance.

[1] The research described in this technical report was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622, N00014-85-K-0124, N00014-91-J-1698 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

Acknowledgments

- Lance Glasser and Dave Gifford, who came through when I needed them.

- Charlie Selvidge and Ellen Spertus, for reading my drafts and not laughing too much.

- Bill Dally, for being helpful in so many ways and pushing me to do my best.

- For my parents, Daniel and Dolores, who by letting me know that it is acceptable to fail, encouraged me to succeed.

- For my other parents, my in-laws James and Lee Moses, for their help in keeping my household together and happy.

- My children, Dan, Kate, and Larry, for having a grouch for a father while he's been doing this.

- Above all, for my wife Judy, for supporting me in this escapade. With her love, I can do anything.

Contents

1 Introduction 7

2 Communication Networks 9
2.1 Topology 9
2.2 Congestion Management 10
2.3 Fault Model 11

3 Reliable End-to-End Protocols 12
3.1 Characteristics of the Protocols 12
3.2 Alternating Bit Protocol 13
3.3 Time Stamps 14
3.4 Windowed Time Stamps 14
3.5 Summary 15

4 The Unique Token Protocol 16
4.1 Overview of the Protocol 16
4.2 Protocol State Diagram 19
4.3 An Example 21
4.4 Proof of Correctness 24
4.5 Performance and Scalability 26
4.6 Summary 28

5 Switching Element Organization 29
5.1 Information Exchange 29
5.2 Block Diagram 31
5.2.1 Input Controller Overview 33
5.2.2 Crossbar Overview 35

5.2.3 Output Controller Overview 35
5.2.4 Router Overview 36
5.3 Summary 37

6 Details of the Switching Element Design 38
6.1 Design Goals and Guidelines 39
6.1.1 Two-Phase Clocks 39
6.2 Top Level Microarchitecture 41
6.3 Timing Generation 42
6.4 Input controller - ict_ict 45
6.4.1 Input Port - ip_ip 46
6.4.2 Flit Checker - ict_check 48
6.4.3 Pointer File - pf_pf 49
6.4.4 Pointer File Load Controller - pf_ctl 50
6.4.5 Dual Ported Ram - dpr_dpr 51
6.4.6 Address Generator - ict_addr 51
6.4.7 Pointer to Address Mapper - ict_pfmap 52
6.4.8 Unload Controller - ict_unload 53
6.4.9 Token Generation - ict_token 56
6.5 Crossbar 57
6.5.1 Token injector - xb_inject 58
6.5.2 Crossbar crosspoints - xbar_elt 59
6.6 Output controller - oct_oct 60
6.6.1 Module I/O 60
6.6.2 Description 61
6.6.3 Output filter - oct_filter 61
6.6.4 Output packager - oct_package 63
6.7 Router 64
6.7.1 Module I/O 64
6.7.2 Description 65
6.8 Summary 66

7 Design Verification 67
7.1 Verification Procedure 67
7.2 Test Jigs and Scaffolding 68
7.3 Testing Results 69
7.4 Summary 70

8 Conclusions 71

A Buffer Storage Requirements of End-to-End Protocols 73

List of Figures

4.1 Packet, token passing 18
4.2 State Diagram for the Reliable Protocol 20
4.3 Simple Network Topology 22

5.1 Switching element for 2D mesh 30
5.2 Frame Format 32
5.3 Switching element block diagram 32
5.4 Input controller block diagram 34
5.5 Crossbar block diagram 35

6.1 Where the nonoverlap time goes 40
6.2 Top Level Module Block Interconnect 43
6.3 Top Level Inter-Module Signalling 44
6.4 Input Controller Block Diagram 47

A.1 Simple Traffic Model 74

Chapter 1

Introduction

Suppose one wanted to construct a building-sized parallel computer. What are the technical obstacles preventing one from doing so? The list is made up of scalability concerns, issues which become increasingly important as the machine gets larger. One key concern is the reliability, which decreases with the size of the computer [10].

The reliability problem can be combatted by introducing a level of fault tolerance into the computer. There are two main methods of providing fault tolerance. The first is static reconfiguration, where the state of the machine is checkpointed regularly to stable storage. Faults cause the machine to halt, a reconfiguration is performed where spares are brought in or portions of the machine are disabled, and the state of the computation is restored from the last checkpoint. Unfortunately, as the machine gets very large and failures quite frequent, the machine spends most of its time checkpointing. It may even reach the point where the machine is unable to reliably checkpoint itself.

The second method of providing fault tolerance is dynamic reconfiguration, where state is saved and restored in local areas of the machine through replication. This allows the machine to continue operation while a fault is occurring. This replication is not without costs. For example, one may replicate the entire communication fabric of the computer to ensure reliable communication between processors, which loses by factor of two in performance to non-reliable machines with equivalent wiring complexity. Alternatively, one may use end-to-end protocols for communications, which require storage in the processors to possibly repeat lost messages and a method of eliminating duplicate messages upon receipt.

We propose in this thesis a third method of achieving reliable communication between processors. This method relies on the use of replicated storage in the communication network and a new protocol, the *Unique Token Protocol.* The protocol requires a constant amount of storage per communication element and hence is no barrier to scaling. It consumes no additional wire bandwidth. Lastly, the overhead for elimination of duplicate messages is only incurred when failures actually happen and thus the overhead becomes essentially negligible.

We will also show the practicality of the unique token protocol by constructing a register transfer level description of a communication element. This model is fairly complex, as one needs an adaptive router to go around possible network faults, and the router must be deadlock free. As a result of the base level of complexity, the model is kept as simple as possible to avoid obscuring the implementation of the reliability protocol.

This thesis is organized into seven main sections. Chapter 2 details the subset of parallel computers for which the reliable protocol was first designed. Properties of the communication networks which impact protocols are given. A simple fault model for those networks is presented. Chapter 3 explores the scalability of end-to-end protocols to very large parallel computers. It demonstrates that the storage requirements for such protocols do not scale linearly with the number of processors. Chapter 4 presents the unique token protocol which uses replication in the network. This protocol is shown to use storage which scales linearly with the size of the machine. Chapter 5 presents an overview of a router designed to support the new protocol. Chapter 6 gives a detailed presentation of the microarchitecture. Chapter 7 describes the design validation which was performed using simulations. Finally, chapter 8 presents conclusions.

Chapter 2

Communication Networks

There exists a large variety of communication networks for parallel computers. The major categories include butterflies [3], shuffle exchanges, hypercubes [1], and meshes [6]. Further permutations exist when one accounts for storage in the network, store-and-forward versus cut-through [12] and wormhole, congestion management policies, etc. All of these have an impact on the protocols used to reliably transport data. In order to focus on the protocol and not the foibles of the network, we restrict ourselves to certain types of networks which have uniform properties. These types of networks are not readily described by the terms "3D mesh" or "hypercube". Instead, we will define what they are in terms of their topologies and congestion management policies.

We are also contrasting *reliable* communication protocols, which implies that portions of the network will sometimes fail. At the end of this chapter, we will describe the network fault model used.

2.1 Topology

An interconnection network is a strongly-connected directed graph, $I = G(N, C)$. The vertices of I are a set of *nodes*, N. N itself is the union of the set of processing nodes P and the set of switching nodes S. The edges are a set of channels, $C = C_{inject} \cup C_{transport} \cup C_{extract}$, where:

$$\begin{aligned} C_{inject} &\subseteq P \times S \\ C_{transport} &\subseteq S \times S \\ C_{extract} &\subseteq S \times P \end{aligned}$$

Each channel is unidirectional and carries data from a source node to a destination node. A bidirectional network is one where $(n1, n2) \in C \rightarrow (n2, n1) \in C$. Since symmetric communication paths between processors simplify the analysis of things such as round trip times, the networks considered here are constrained to be bidirectional.

The network must be able to maintain connectivity in the event of a fault. The network is thus required to have at least two disjoint paths between any two switching nodes. The paths must be disjoint in terms of the switching elements appearing on each path.

Many topologys satisfy these requirements. They include toruses, hypercubes, meshes, and express cubes [8]. Later, examples of the protocols will be given using two-dimensional mesh networks. This is to facilitate the presentation and should not be construed as a binding requirement.

2.2 Congestion Management

Using the unique token protocol, the network promises to always deliver the packet. The network is not allowed to drop packets, which implies that the network must have storage for packets in the event of congestion. A switching node may provide storage for an entire packet, as in the case of store-and-forward networks. Alternatively, only portions of a packet may be stored at each switching node, as in virtual cut-through and worm-hole routing. The choice of buffering strategy does not affect the unique token protocol.

The network must have some facility for routing around failed channels and/or switching nodes. For example, dimension-ordered routing in a 3D-mesh network is not tolerant of any minor irregularities in the topology. Adaptive routing algorithms such as those using virtual channels or chaotic methods are required [4, 5, 14, 11]. Further, such algorithms must either be deadlock free or provide a method other than packet dropping to break the deadlock [9].

2.3 Fault Model

The fault model used throughout this thesis is a simple one. We assume that failures occur at single points. We further require that the time between failures is sufficiently large to allow the network to heal.

For wires, the fault model is a stuck-at fault. When a wire fault is detected, the fault is assumed to be permanent, not transient. This is done because the process of deciding whether or not a wire failure is transient is technology dependent. For example, a wire failure between two adjacent chips on a PC board is usually taken as indication that something is physically wrong with the board. If the wire is actually a cross country link using commercial phone lines, one is more apt to blame noise for the failure.

Nodes are required to operate in a fail-stop fashion for much the same reason, in that there does not exist a good technology or implementation independent model of node failures. The unique token protocol is not designed to be robust in the presence of Byzantine failures. If one did not impose the fail-stop requirement, one would always able to contruct some failure mechanism which causes a particular implementation of the protocol to also fail.

Lastly, we require that failures not occur too closely to one another. This is to avoid multiple single point failures from appearing to be multiple point failures.

Chapter 3

Reliable End-to-End Protocols

We have already seen that large scale parallel computers require reliable interprocessor communication. In this chapter, we will look at the problem a reliable protocol tries to solve and then explore several end-to-end protocols. We will see that while these protocols do solve the reliability problem, they do not scale linearly with the size of the machine. We measure machine size by the number of processors and represent the size of the machine by the letter n.

3.1 Characteristics of the Protocols

An entity on node A wishes to send a message to an entity on node B. We wish that the entity on node B processes the message exactly once. A subtle point is that is perfectly acceptable for node B to receive many copies of the same message, as long they are processed exactly once. [1]

If a portion of the network which holds the message as it transits from node A to node B could fail, then the message needs to be replicated somewhere. It could be replicated in the network, at node A, or both. Where the

[1] This may be loosened even further to allow the multiple processing of the same message, as long as the processing is idempotent. We believe that most messages in parallel computers result in actions which are not idempotent, so we are unable to take advantage of idempotence.

the replication occurs is a key property of the protocols.

With multiple copies of the same message and imperfect communication between any two copies, it is possible to deliver duplicates of a message to node B. The destination must now discard all duplicates and process the message exactly once. Thus, the other key property of a protocol is how the destination detects and discards redundant messages.

3.2 Alternating Bit Protocol

The alternating bit protocol [2, 16, 19] relies on a FIFO communication channel between two nodes, A and B. The channel is bidirectional. The alternating bit is a reusable sequence number ranging over 0 and 1.

The protocol begins with node A sending a message to node B. The message contains the sequence number 0. Node A repeatedly sends the message until it receives an acknowledgment having a sequence number of 0. Node A then shifts to sequence number 1 and starts its next message. Node B sends acknowledgments each time it successfully receives a message. Node B only processes messages whose sequence number differed from the last sequence number successfully received.

This protocol cannot readily be used in a parallel processor for the following reasons:

1. It does not allow multiple, unacknowledged messages to be in flight between the two nodes. This is a severe performance bottleneck.
2. It relies on FIFO channels. This is certainly not the case in a network with adaptive routing.
3. The sequence number must be maintained between each pair of nodes. This scales as $O(n^2)$, which is not acceptable for large machines.

3.3 Time Stamps

Time stamps are a way of augmenting the alternating bit protocol to allow multiple messages in flight and account for out-of-order message delivery. When node A first decides to send a message to node B, it stamps it with the current time. The message is then sent to node B.

When node B receives the message, it checks to see if this message is newer than all previous messages from node A. If it is, node B processes the message and returns a positive acknowledgment to node A. If the message was not newer, a negative acknowledgment is returned.

If node A receives a positive acknowledgment, it is done with that message. If the node A receives a negative acknowledgment for a message it believes to have been sent successfully, the negative acknowledgment is ignored. If it receives a negative acknowledgment for a message which it believes hasn't been successful, it updates the time stamp on the message and resends the message. Lastly, node A may time out and simply resend the message.

This protocol can be used in a parallel processor, but it still suffers from the following problems:

1. When messages arrive at the destination out of order, they will need to be retransmitted.

2. The sequence number must be maintained between each pair of nodes. This scales as $O(n^2)$, which is not acceptable for large machines.

3. Choosing the size of the timeout window is difficult when the network has large variations in latency.

3.4 Windowed Time Stamps

This protocol is basically the same as the time stamp protocol, except that the destination remembers the n newest time stamps it has seen from a source. This allows the destination to accept messages which arrive slightly out of order. The protocol only suffers from:

1. The size of time stamp tables in the destination do not scale.
2. Choosing the size of the timeout window is difficult when the network has large variations in latency.

3.5 Summary

We have explored three different end-to-end protocols. We have seen that the expected space requirements of the protocols are not constant per node, owing to the n^2 growth in the destination time stamp tables and the $\sqrt{n}$ growth of the source buffer pools[2]. Further, determining "good" source buffer-pool sizes and time-out intervals is largely empirical. These parameters may not yield satisfactory performance when new programs offer traffic not in accordance with the simulation model. We are thus motivated to look for new protocols, which is the subject of the next chapter.

[2] A derivation is contained in Appendix A

Chapter 4

The Unique Token Protocol

We saw that the end-to-end protocols replicate packets in the source, requiring storage in amounts that do not scale linearly with the size of the machine. The destination requires a large table to eliminate duplicates which also does not scale. As alluded to in the chapter on end-to-end protocols, there are several places in a system where packets can be replicated and many ways for the destinati:n to eliminate duplicates. In this chapter, we propose an alternative reliable protocol which uses the network to replicate packets. This protocol, the unique token protocol, will be shown to have both satisfactory performance and implementation costs.

4.1 Overview of the Protocol

We observe that many networks already provide storage for packets in flight. We propose to use that storage for maintaining duplicate copies of a packet in the network, rather than keeping copies at the source. By distributing the storage throughout the network, we will be able to bound the storage to a constant amount per node.

We also noticed that the end-to-end protocols required large tables to handle possible packet duplication, even though most the time duplicates were not sent. It would be desirable if, when the network delivers multiple copies[1] of a packet, all copies of the packet were marked as being copies. Thus, when a packet arrived marked as unique, the destination node would not have to worry about duplicate elimination. Only in the rare case of actual errors would the duplicate elimination machinery be used.

We begin the presentation of the unique token protocol with a model of a source node sending a packet to a destination node, where the packet is buffered and forwarded at several intervening switching nodes along the way. The buffering and forwarding process needs to ensure that at least two copies of the packet exist along the path from source to destination at all times. This is done by first copying the packet forward one node, then allowing the release of the storage in the rearmost node. This is shown in figure 4.1. Note that any node along the path only receives one copy of the packet, even though multiple copies are kept.

When the packet first enters the network, a token is sent along behind the packet. We will use the token to determine when the packet has been duplicated in the network by preserving an invariant, that no copies of the packet exist in the path behind the token. Thus, if a single path is used to connect source and destination, the arrival of a token at the destination implies that the packet has been delivered exactly once.

When a portion of the network fails, communication between the advance and rear copies of the packet may be severed. Neither copy knows the fate of the other, so they both must make their way to the destination. When they arrive at the destination, both packets must be marked in such a way as to cause the destination to look for duplicates. The protocol does this by establishing two types of tokens: *unique* or *replica*. When the token is first sent along behind the packet, the type is unique. If the network *ever* has to use multiple paths while forwarding a packet, the token is changed to type replica for all copies of the packet.

The conversion of tokens of type unique to tokens of type replica happens when the network fails. The rear copy of the packet is still on the path leading from the source and will have a token coming up from behind. The rear copy simply chooses a new forward path to the destination and copies itself forward. When the token arrives at the point of reroute, it is converted

[1] "copies" includes the original, plus all replicas made in the event of the loss of the original.

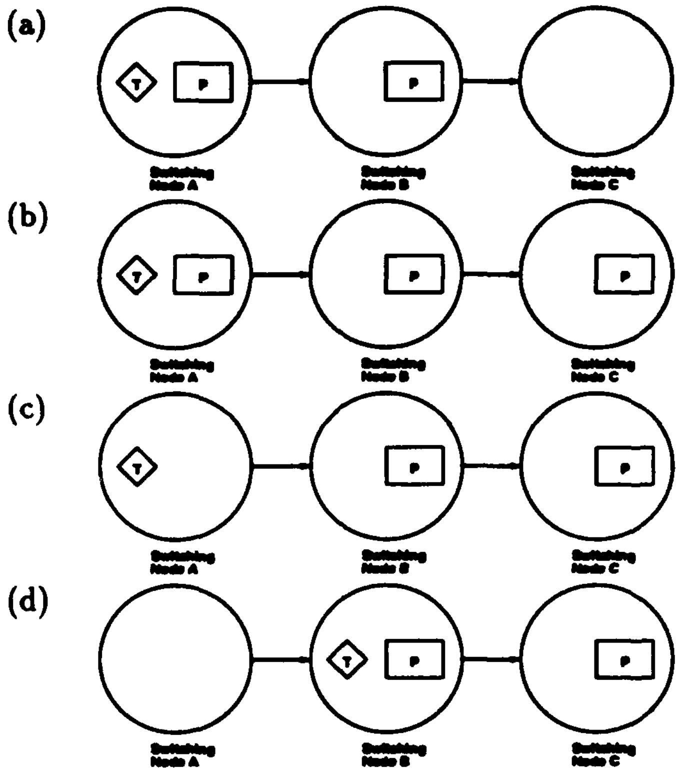


Figure 4.1: An illustration of packets and the token moving through the network. (a) Switching node A has the token and a copy of the packet, switching node B has a copy of the packet, and switching node C is empty. The token-passing portion of the protocol begins with the nodes in this state. (b) Switching node B then copies its packet to node C. (c) When node B has successfully copied its packet to node C, node B sends an acknowledgment to node A. Node A now knows that the packet is safely in two other places, and is able to delete its copy. (d) The last remaining step is for node A to pass the token to node B.

to replica. The advance copy lost the path back to the source and will not see a token. The advance copy generates a new token of type replica and continues on to the destination. We rely on packets containing some form of unique identifier to allow the destination to eliminate the duplicates.

4.2 Protocol State Diagram

The full protocol used by a single node for moving packets and tokens is shown in figure 4.2. There are 12 states in total. States 1–6 are used under normal operation, states 7–12 are used when errors occur. In the detailed description that follows, there are actually three nodes involved. There is the node whose state transitions are being displayed (simply the node), the node closer to the source (the upstream node), and the node closer to the destination (the downstream node).

The inputs to the state machine are tokens, packets, and acknowledgments. Tokens and packets are placed in one first-in-first-out (FIFO) queue, acknowledgments are placed in a second. This allows the state machine to disregard the order of arrival of these events. In particular, this simplifies the handling of the arrival of a second packet while the node is still forwarding a token.

(1) **Idle** The idle state is left whenever a packet arrives from the upstream node.

(2) **Forward Packet** The node determines a route for the packet and forwards the packet to the downstream node. The node waits a short, fixed period of time to allow the downstream node to signal the success or failure of the transfer.

(3) **Send Ack** The node sends an acknowledgment to the upstream node, indicating that three copies of the packet now exist and that it is safe for the upstream node to erase its copy.

(4) **Wait for Ack** The node now waits for the downstream node to tell it that its copy of the packet may be erased.

(5) **Wait for Token** The node erases its copy of the packets and waits for the token from upstream.

(6) **Forward Token** The token is sent downstream and the automaton returns to idle.

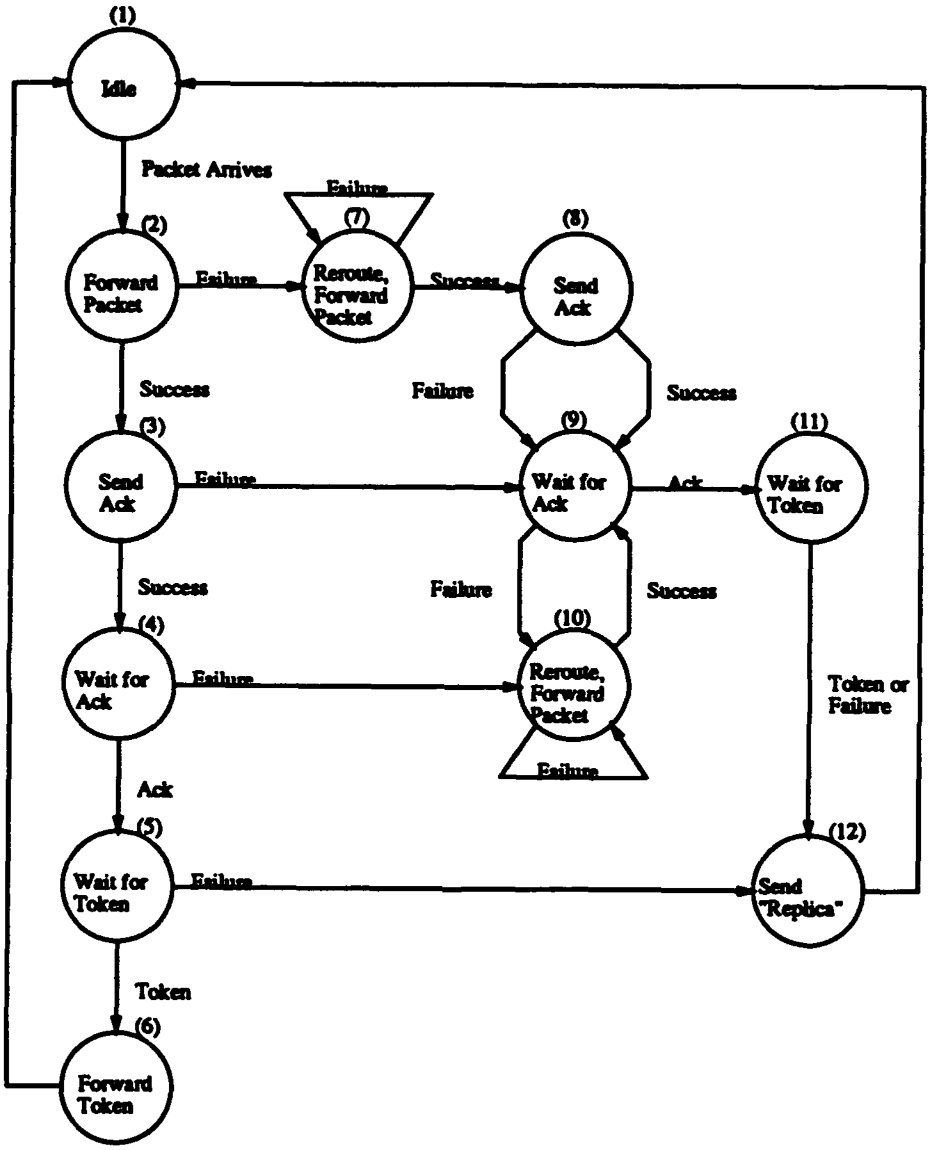


Figure 4.2: State Diagram for the Reliable Protocol

(7) **Reroute** The packet is rerouted and forwarded. This process repeats until succeeding.

(8) **Send Ack** The node sends an acknowledgment to the upstream node,

indicating that three copies of the packet now exist and that it is safe for the upstream node to erase its copy. Since this state is entered under error conditions, the success or failure of the acknowledgment transmission does not matter.

(9) **Wait for Ack** The node now waits for the downstream node to tell it that its copy of the packet may now be erased.

(10) **Reroute** The packet is rerouted and forwarded. This process repeats until succeeding. Since an acknowledgment has already been sent upstream, this state returns to the wait-for-ack state (9).

(11) **Wait for Token** The node erases its copy of the packets and waits for the token from upstream.

(12) **Send Replica** This state is entered only after an error has occurred, so the token sent downstream must be of type replica.

4.3 An Example

We now give examples of the protocol moving a packet through a simple network topology, as shown in figure 4.3. Node A has generated a tagged packet which is destined for node B.

The tagged packet must travel via either switching elements SE1,SE2,SE4 or SE1,SE3,SE4. We will assume that the preferred path is SE1,SE2,SE4. Here is the protocol, given that no errors occur.

1. The protocol begins with node A holding a tagged unique packet. Node A first transmits the packet part of the tagged packet to SE1.

2. SE1 sends the packet part to SE2 while keeping a copy.

3. When SE1 determines that the packet part has been successfully transferred to SE2, SE1 sends an acknowledgment to Node A.

4. When node A receives the acknowledgment, it knows that the packet is now safely stored in two places in the network: SE1 and SE2. Node A then deletes its copy of the packet and sends the token to SE1.

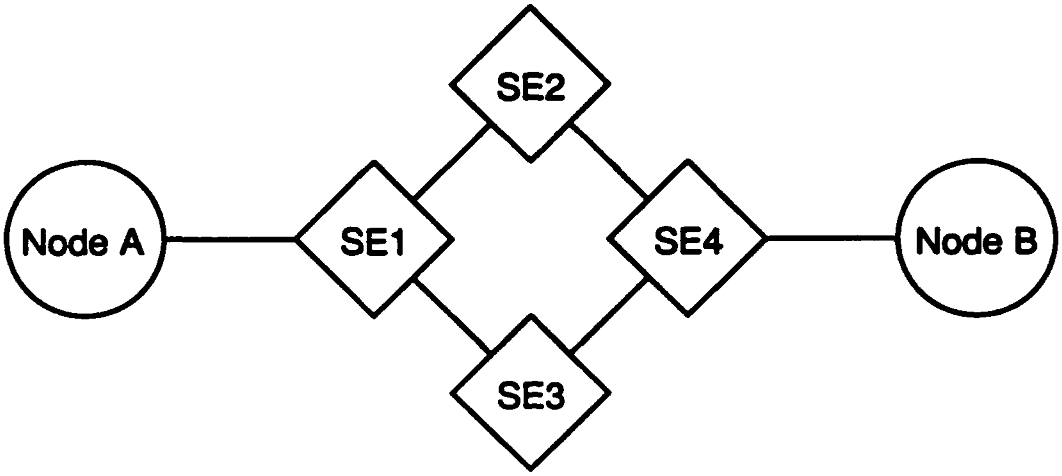


Figure 4.3: Simple Network Topology

5. Meanwhile, SE2 has transmitted the packet to SE4 and sent an acknowledgment to SE1.
6. When SEl has both the acknowledgment and the token, it deletes its copy of the packet and sends the token to SE2.
7. SE4 sends the packet to node B and sends an acknowledgment to SE2.
8. SE2 now has the token and an acknowledgment, so it deletes the copy and sends the token to SE4.
9. Node B, being the final destination of the tagged packet, sends an acknowledgment to SE4 as soon as it receives the packet portion.
10. SE4 deletes its copy and sends the token to node B.

Node B now has a tagged unique packet and we see that no copies exist in either node A or any of the switching elements. The tagged unique packet has been delivered exactly once.

We now need to examine what the protocol does when errors occur. In this example, the fault will occur in the channel between SEl and SE2.

1. The protocol begins with node A holding a tagged unique packet. Node A first transmits the packet part of the tagged packet to SE1.

2. SE1 sends the packet part to SE2 while keeping a copy.

3. When SE1 determines that the packet part has been successfully transferred to SE2, SE1 sends an acknowledgment to Node A.

4. When node A receives the acknowledgment, it knows that the packet is now safely stored in two places in the network: SE1 and SE2. Node A then deletes its copy of the packet and sends the token to SE1.

5. SE2 now transmits the packet to SE4 and tries to send an acknowledgment to SE1, but the channel between SE1 and SE2 goes down. SE2 manufactures a *replica* token and proceeds as if the token was received from SE1.

6. SE4 sends the packet to node B and sends an acknowledgment to SE2.

7. SE2 now has a replica token and an acknowledgment, so it deletes the copy and sends the replica token to SE4.

8. Node B, being the final destination of the tagged packet, sends an acknowledgment to SE4 as soon as it receives the packet portion.

9. SE4 deletes its copy and sends the replica token to node B.

10. Node B now has a tagged replica packet. Node B extracts the UID contained in the packet and compares it against the list of replicas it has received so far. Since it has not been seen before, node B adds it to the list and processes the packet.

11. Meanwhile, SE1 notices that the channel between SE1 and SE2 has gone down. It changes its token from unique to replica. It then takes its copy of the packet and sends it to SE3.

12. SE3 transmits the packet to SE4 and sends an acknowledgment to SE1.

13. When SE1 has both the acknowledgment and the token, it deletes its copy of the packet and sends the replica token to SE3.

14. SE4 sends the packet to node B and sends an acknowledgment to SE3.

15. SE3 now has the token and an acknowledgment, so it deletes the copy and sends the token to SE4.

16. Node B, being the final destination of the tagged packet, sends an acknowledgment to SE4 as soon as it receives the packet portion.

17. SE4 deletes its copy and sends the replica token to node B. Node B now has a tagged replica packet, so it extracts the UID and checks its list. Since the UID matches, the packet is discarded.

We see that although node B has received two copies, it processes only one. Thus, for this simple example, the protocol works.

4.4 Proof of Correctness

The example given shows that the protocol works correctly in a select case. We will now show that the protocol is correct in general. The correctness properties we will show are:

- a copy of the packet is always delivered,
- a packet is always delivered with exactly one token, and
- if a packet is delivered tagged with a unique token, then it is the only only copy of that packet delivered.

We begin by restating our assumptions about the network. We require that the network remain fully connected after all faults have occurred, to prevent stranding a packet in the network. The router must be able to route around any faults. The faults are single points of failure and do not occur too closely together in time. Lastly, we require that progress delivering tokens or packets is always being made.

Theorem 1 *If a a node s sends a packet to destination t, a copy of the packet is always delivered.*

Proof The unique token protocol keeps at least two copies of any packet in the network. This was illustrated in figure 4.1, where the packet is copied forward before the rearmost copy is erased. The fault model allows the destruction of at most one copy per fault. After the fault has been detected and the packet started on a new path, the packet is again duplicated. Thus a packet will always be delivered. □

The next two properties are difficult to prove, as the protocol relies on implicit channel state to bind a token and a packet. We will make that binding explicit by augmenting the token with the destination of the packet and the packet's unique identifier. This restriction requires that a node actually have a copy of the packet before it is able to generate a token. This does not violate the spirit of the protocol, as a node need only erase its copy of the packet before transmission of a token and we observe that the protocol will only send tokens after it received a packet.

The second property requires two lemmas, to show that at most one copy and at least one copy are delivered.

Lemma 2 *A node receives at most one copy of a token for each packet it receives.*

Proof In order for a token to be delivered, it must have been sent from another node. In order for the token to be sent, the protocol must have passed either the *forward token* or *send replica* states shown in figure 4.2. In order to reach either of these states, a copy of the packet must have been sent. Once the token is sent, the packet and the token are erased. It is not possible to generate additional copies, as the unique identifier is lost. □

Lemma 3 *A node receives at least one token for each packet it receives.*

Proof If a node receives a packet on a channel and communication is not subsequently severed, the node will eventually receive a token, owing to the network's progress property. If communication is severed on the channel before it receives a token, the node will generate one new token and bind it to the packet, as shown in the *wait for token* and *send replica* states of figure 4.2. □

Theorem 4 *A packet is received with exactly one token.*

Proof We combine Lemma 2 and Lemma 3. □

Lemma 5 *If a unique token is transmitted from a node, only one copy of a packet was transmitted from that node.*

Proof Tokens are only transmitted in states *forward token* and *send replica*. In order for a unique token to be transmitted, the protocol must have passed through the *forward token* state. There is only one set of state transitions which allow the protocol to pass through that state. On that set of transitions, the packet is transmitted exactly once in state *forward packet*. □

Theorem 6 *If a packet tagged with a unique token is delivered to a destination, then it is the only copy of that packet delivered.*

Proof By contradiction. Suppose that multiple copies of a packet were delivered to the destination, along with a unique token on one of the packets. In order for multiple copies of packet to be delivered, a node between the source and destination must have transmitted multiple copies. At the node where multiple packets were transmitted, none of the transmitted packets were tagged unique. As those packets found their way to the destination, none of the tokens could have been converted to type unique, so no unique tokens could have been delivered. □

4.5 Performance and Scalability

Under the unique token protocol, assuming error-free operation, each node transmits a packet, an acknowledgment, and a token. The token is quite small and can be represented by a single bit. The acknowledgment is only used between nodes and could be represented in a few bits. Thus, the bandwidth between nodes is used primarily for moving the packet. The end-to-end protocol required that the acknowledgment be sent end-to-end, consuming bandwidth equal to that of a small packet. Therefore, in terms of wire bandwidth, the unique token protocol is superior.

The unique token protocol was designed to eliminate the nonlinear growth in storage of end-to-end protocols. There are two storage components of concern: the storage in the source and the duplicate elimination tables. The unique token protocol eliminates the storage in the source. Instead, it uses storage in the network. From the discussion of packet replication, we see that the network requires buffers for at most three copies of any packet. This compares well to the two copy requirement for ordinary store and forward (copy forward, then erase). When wormhole routing is used, the buffering requirements approach two copies for the unique token protocol and one copy for the end-to-end protocols. The buffering required is a fixed amount per switching node, which implies that the amount of memory in the system dedicated to packet replication grows linearly with the size of the machine.

Duplicate elimination must still occasionally be performed at the destination, requiring tables. However, the tables need only log packets whose tokens are of type replica, not those tagged as unique. Under error-free conditions, the log will be empty. When an error occurs, only a fixed number of replicas can be created before the network heals and packets are adaptively routed around the fault. Restated, faults only cause replica of packets to be made when those packets were near the fault around the time of failure, not as a result of a static adaptation. If faults occur one at a time, only a fixed number of replicas can ever be sent to a destination, allowing the duplicate elimination log to be of fixed size. Again, a fixed size per destination node allows a linear growth in the memory requirements of the system.

In terms of latency, the protocol does not delay the arrival of the packet at the destination, even if the token is slowed. One simply winds up with an excess number of copies of the packet distributed along the path between the token and the destination. However, since the packet arrives at the destination while the token is at least two nodes away, there is approximately a two hop delay before the destination knows the type of token.

This need not be an impediment to the start of processing the packet. If one is able to bound the maximum packet latency in the system, one is able to timestamp the entries in the duplicate elimination log and discard them after all possible copies have been received owing to the time constraint.

Since we expect the maximum delivery time to be small, say on the order of milliseconds, and the interarrival time of failures to be large, say on the order of tens of seconds, most of the time the log will be empty. When the log is empty, we can safely process any packet which arrives, as long as the tokens for all previous packets have been seen.

Of course, if several packets arrive before their tokens, due to the time multiplexing of several packets onto the same physical channel, one is always able to do a small amount of ordinary duplicate elimination as in the end-to-end protocols. The size of the table required for this corresponds exactly to the number of packets time-multiplexed onto the physical channel. Again, these are all fixed size tables, so the overall storage requirement grows linearly with the size of the system.

4.6 Summary

In this chapter, we described a protocol which will reliably deliver messages without the use of an end-to-end protocol. The protocol's correctness was demonstrated in an example and then proven correct. The new protocol only requires constant amounts of storage in each node to allow replication of packets. Thus, the protocol imposes no limits on machine scaling owing to memory constraints. The protocol was shown to be more wire bandwidth efficient than end-to-end protocols. It was also shown not to increase the latency from the start of message transmission to the start of message processing. On all three critical measures: memory, bandwidth, and latency, the unique token protocol is superior or equal to end-to-end protocols.

Chapter 5

Switching Element Organization

In the preceding chapter we presented the unique token protocol which is to be implemented in the network. We need to understand the costs of such an implementation. In developing the organization of the switching element, we will gauge its complexity and look for unforeseen costs or benefits. We choose 2D meshes as the base network, as they afford simple non-reliable implementations and the complexity of a reliable version will be due mainly to the reliability features. Similarly, we organize the parts of the design for clarity, not for performance, as we want the design to be intuitively correct. This chapter will describe how a reliable switching element intended for use in 2D-mesh networks may be organized.

5.1 Information Exchange

The switching element has interfaces to its four geographic neighbors and its processor, as shown in figure 5.1. Here, the same type of interface is used to connect the switching element to the processor as is used to connect to the neighboring switching elements, although it need not be the case in other designs. To form a network, the switching elements are tiled into a plane and are assigned (x, y) coordinates.

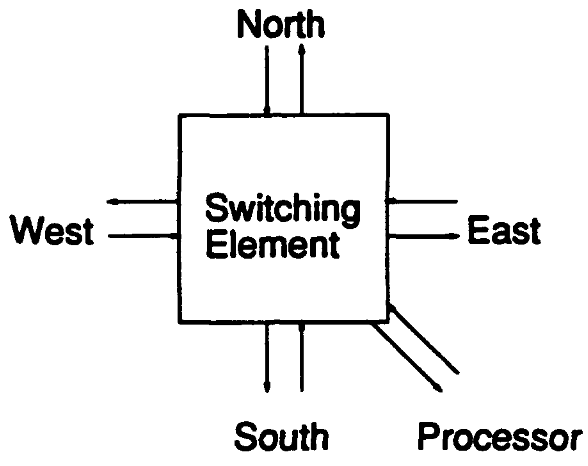


Figure 5.1: Switching element for 2D mesh

To describe how these switching elements communicate, we will use the terminology suggested by Dally [7]. Communication between processing elements is performed by sending *messages* through the network. Messages may be arbitrarily long, so a message may have to broken into one or more *packets* for transmission. Packets contain sequencing information to allow them to be reassembled into a message. They also contain routing information and are the smallest unit of information exchange to do so. The routing information is specified as a destination (x, y) coordinate pair. Packets must carry a unique identifier for possible duplicate elimination at the destination. This unique identifier is simply the source's (x, y) coordinate pair concatenated with a small sequence number.

A packet contains three or more flow control digits or *flits*. They are the smallest unit on which flow control is performed. By decomposing the packets into flits, the packet may be wormhole routed over virtual channels. Flits from multiple packets are time division multiplexed onto a single physical channel, forming the virtual channels. The virtual channels associated with a single physical channel share physical bandwidth, allocated on a flit by flit basis. Since the flits must be demultiplexed when received, they carry a virtual channel identifier.

Flits come in three main kinds: *head*, *data*, and *token*. A packet is made of one head flit, one or more data flits, and one token. The head flit contains the routing information. The first data flit carries the unique identifier. Subsequent data flits are used for the actual data transport. The token flit cannot carry any data and comes in two sub-kinds: *unique* and *replica*. Tokens of sub-kind replica are generated during network failures.

Flits carry sequence numbers to allow their reassembly into a packet in the event that that network failure cuts the packet in two. This reassembly information could be maintained as a starting-flit-number field in the head flit of the packet. It is not done here in order to simplify the design.

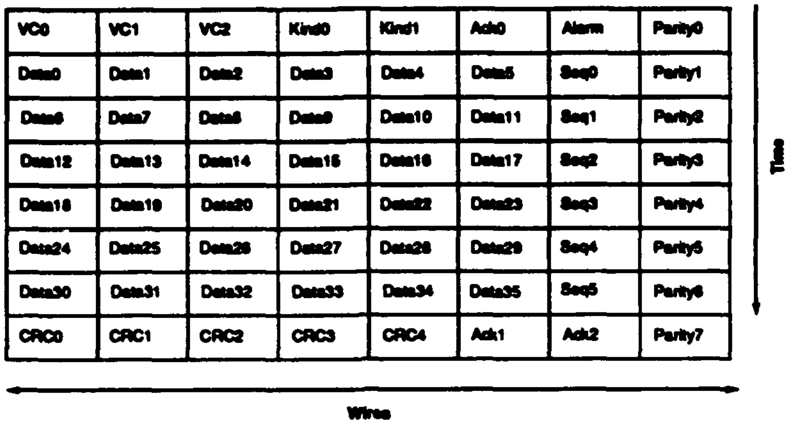
Flits are transferred over physical channels in physical transfer units or *phits*. A phit is usually smaller than a flit as one encounters pin-count restrictions on actual IC devices. For example, 36 data bits over 7 bidirectional channels would require 504 signalling pins. In this particular case, we wanted to keep the total pin-count for the switching element in the neighborhood of 100 pins. With 5 ports, this came out to be under 10 pins per phit, so a width of 8 bits was chosen for a phit. One of the 8 bits is used for parity, so 7 remain for the movement of data between devices.

At least 6 phits are needed to transport a data field of 36 bits. After counting 11 additional bits for virtual channel numbers, sequence numbers, and kind fields, we see that 7 phits are nearly fully used in transporting a flit. Other information, such as acknowledgments and alarms must be communicated between devices. To make efficient use of the wires between devices, we use the same wires as are used to in transporting the flit and time multiplex onto them this additional information. The time multiplexing is done in a fixed manner and we call the structure of the time multiplexing a *frame*. Figure 5.2 shows the complete frame format.

5.2 Block Diagram

The switching element, as shown in figure 5.3, is composed of four types of elements. They are input controllers, a switch, output controllers, and a router. Before delving into the details of each module, a few top level items need to be explained.

The first item is where to put the the buffers. Since the switching element does not drop flits under contention, buffering for flits must be provided. In



VC0	VC1	VC2	Kind0	Kind1	Ack0	Alarm	Parity0
Data0	Data1	Data2	Data3	Data4	Data5	Seq0	Parity1
Data6	Data7	Data8	Data9	Data10	Data11	Seq1	Parity2
Data12	Data13	Data14	Data15	Data16	Data17	Seq2	Parity3
Data18	Data19	Data20	Data21	Data22	Data23	Seq3	Parity4
Data24	Data25	Data26	Data27	Data28	Data29	Seq4	Parity5
Data30	Data31	Data32	Data33	Data34	Data35	Seq5	Parity6
CRC0	CRC1	CRC2	CRC3	CRC4	Ack1	Ack2	Parity7

Figure 5.2: Frame Format

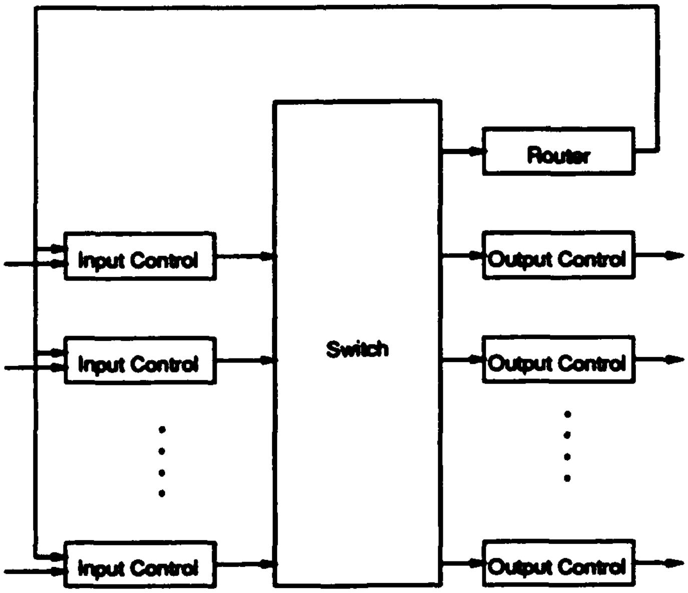


Figure 5.3: Switching element block diagram

this organization, the buffering was chosen to be solely in the input controller. This was done for several reasons. The first is that the presence of output queueing implies that the peak transfer rate out of the switch is higher than the peak transfer rate of the physical channel, which requires wider internal data paths or higher internal clock rate. Both of these seem unlikely in practical applications. The second reason has to do with the generation of acknowledgments. The switching element cannot send an acknowledgment upstream for a flit until that flit has successfully been copied downstream. If queueing is done in the output controller, multiple output controllers may decide to send acknowledgments over the same physical channel at the same time, contending for use of the acknowledgment signals. Solutions involve either separate acknowledgment signals for each virtual channel or some form of queueing on acknowledgments, both of which appear to be impractical.

The second item deserving explanation at this point is the use of a single centralized router. The motivation behind this is that the router allocates resources, the output virtual channels. If the router were to be distributed among the input controllers, some form of arbitration for those resources would have be implemented. Although distributing the router is desirable to avoid queueing delay for the centralized router, the arbitration for virtual channels and communication of deadlock avoidance information is complex. Attempting to describe such a distributed router is deemed outside the scope of this thesis.

The last item is that the entire switching element is flit synchronous. This was done to avoid the possibility of livelock arising from having control loops longer than the fundamental unit of message synchronization. If the switching element were to operate phit-synchronously, one would have to go through all control loops and guarantee that livelock could not arise. It is far simpler to make the unit of synchronization longer than any control loop and suffer a performance loss.

5.2.1 Input Controller Overview

The input controller, shown in figure 5.4, logically consists of four submodules: a loader, a dual-ported RAM, a pointer file, and an unloader. The loader places flits into the dual-ported RAM. The unloader is responsible for

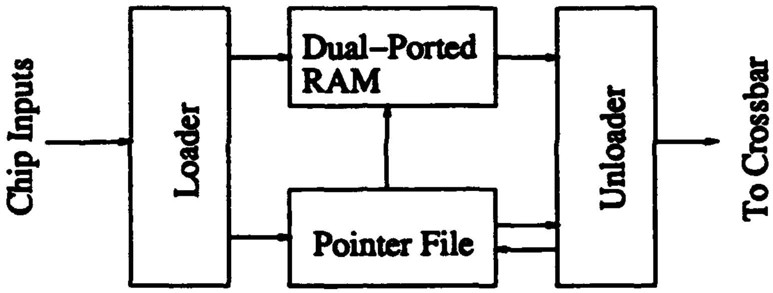


Figure 5.4: Input controller block diagram

maintaining all channel state, interacting with the router and output controllers, and removing flits from the RAM. The pointer file keeps all the read/write pointers into the RAM.

The loader contains modules to receive phits from the physical channel, check the parity and CRC fields, and place flits into the dual ported RAM.

The loader communicates the virtual channel state information to the unloader via a pointer file. The pointer file maintains both write and read pointers into the RAM for all virtual channels. The pointer file also keeps a count of the number of acknowledgments sent for each virtual channel.

The unloader assigns one of three states to each input virtual channel: *idle*, *waitingForRoute*, or *active*. The unloader monitors the pointer file, looking for channels with unsent flits. This is determined by a disparity in read and write pointers. If the channel is idle, the first flit is sent to the router and the channel state changed to waitingForRoute. If the channel is active, the flit is forwarded to correct output port. If the channel is in the waitingForRoute state, nothing is done.

When the router establishes a route, it changes the state of the virtual channel to active. If the route must be retried, the state is changed to idle.

The unloader contains the logic for generating acknowledgments. An acknowledgment must be delayed a few flit times after the flit enters the crossbar, to allow the error checking on the output link to take place.

The unloader also contains logic for changing flits of type tokenUnique to type tokenReplica, as well as being able to generate a flit of type tokenReplica.

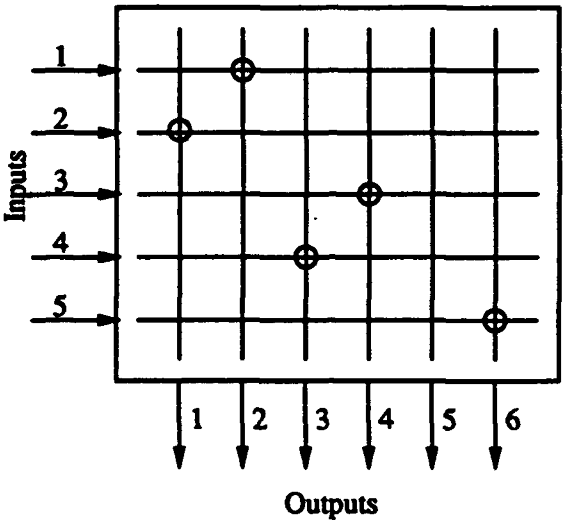


Figure 5.5: A crossbar. Input 1 is connected to output 2, 2 to 1, 3 to 4, 4 to 3, and 5 to 6.

5.2.2 Crossbar Overview

The crossbar moves flits from input controllers to output controllers and the router, which results in an asymmetric crossbar. The input controller submits a bid to the crossbar, indicating to which output it wishes to send a flit. The crossbar responds with a bid-accepted signal if the crosspoint is established. The output controller is told of the crosspoint by the column valid signal. A typical crossbar is shown in figure 5.5.

The output controller or router indicates acceptance of a flit by asserting the destination-accepts signal. The output controller may refuse a flit if transmission of the flit would overflow the downstream input buffer. In addition, the crossbar provides an indicator of output channel health back to the input port.

5.2.3 Output Controller Overview

The output controller is composed of two submodules, the output filter and the output packager. The output filter maps input <port,virtual channel>

tuples to output virtual channels. The virtual channel field in the flit is altered in this submodule. The output filter also implements virtual channel flow control by not accepting a flit from the crossbar when sending that flit would cause a buffer to overflow.

The output packager computes and affixes parity for the first and last phits of a flit. Parity for the intermediate phits are end-to-end. In addition, the output packager computes and affixes the CRC field.

5.2.4 Router Overview

The router is responsible for the mapping of input <port,virtual channel> tuples to output ports. The router receives head flits from the crossbar. The router disassembles the flit into a dimension reversal (DR) number and destination x, y, and z coordinates. The DR number is broadcast to all output controllers. Each output controller responds with three pieces of information:

- if the output port is up,
- if there are any free virtual channels, and
- if there are no free channels, whether or not the packet can wait for a virtual channel to free up.

The router then attempts to perform dimension ordered routing. If the desired output controller is down, an output controller is selected at random. If the targeted output has no free channels and the packet cannot wait due to DR number constraints, a new output controller is picked.

If the targeted output controller has no free virtual channel, the router tells the input controller to retry the route request. If there is a free virtual channel, the router signals the output controller to reserve a virtual channel at a given DR level. The router also notifies the input controller that a route has been established and that the input controller may proceed with forwarding the packet.

5.3 Summary

In this chapter, we presented the top-level organization of a switching element implementing the unique token protocol. The communication between switching elements was described. A block diagram of the switching element was explained and several architectural motivations were given. Details of the four top-level blocks were then filled in. We noted that this particular organization was chosen to minimize the design complexity of the microarchitecture, which is the subject of the next chapter.

Chapter 6

Details of the Switching Element Design

The low-level architecture and design must look ahead to the physical implementation. The easiest implementations (for humans) using our CAD tools are done using standard cells. This implies that the logic complexity cannot be too great, nor can more than a few specialized structures be built. Standard cell designs are also difficult to optimize for speed, so we do not attempt to push the library cells to their limits. We also restrict ourselves to a clock methodology and a sequential logic style which may be readily verified using a timing analyzer such as Crystal.

Design guidelines are specified to support the design goals. The guidelines cover the clocking methodology and how hand timing calculations may be done. They discuss documentation style, the use of custom circuits and circuit conventions.

A detailed architecture for the switching element is presented. It is given at the module level, which should correspond to a page or two of schematics per module. For each module, we list its inputs and outputs and give a description of the functionality contained in the module. The signal names used in the descriptions are intended to be the same as those used in the M-language[1] model of this architecture, found in the appendix.

[1] Mentor Graphics, Silicon Design Division

6.1 Design Goals and Guidelines

The primary design goal is to demonstrate the logical correctness of the unique token protocol under operating conditions. The protocol is built on top of a full-featured router, one that is both adaptive and provably deadlock free. This design base is non-trivial and we try to simplify the design wherever possible. We therefore minimize the number of basic mechanisms and do not perform some obvious optimizations. For example, portions of the router could be distributed among the input controllers, optimized for packets traveling in straight line. This is a common case and would eliminate congestion at the router module. However, it would add considerable logic complexity and chip area, so it is deemed to be not worth the risk.

6.1.1 Two-Phase Clocks

The chip is synchronous and derives all timing from a two-phase non-overlapping clock. An single external clock is supplied and the clock buffer logic will internally generate the two non-overlapping phases PH1 and PH2. The two high-true phases PH1 and PH2 are distributed across the entire switching element. The low-true versions PH1n and PH2n must be derived locally in each module.

For the MOSIS 2μ process, τ is about 75ps. Our goal is a clock period of 600τ, 200τ per phase with 100τ of non-overlap. Figure 6.1 illustrates where the nonoverlap time of 100τgoes. Intermodule clock skew is $+/- 25\tau$. For SPICE, we use slow-N, slow-P, at 50C and 4.5v.

Strict two-phase discipline is used throughout the chip. All signals are labeled with their type, one of:

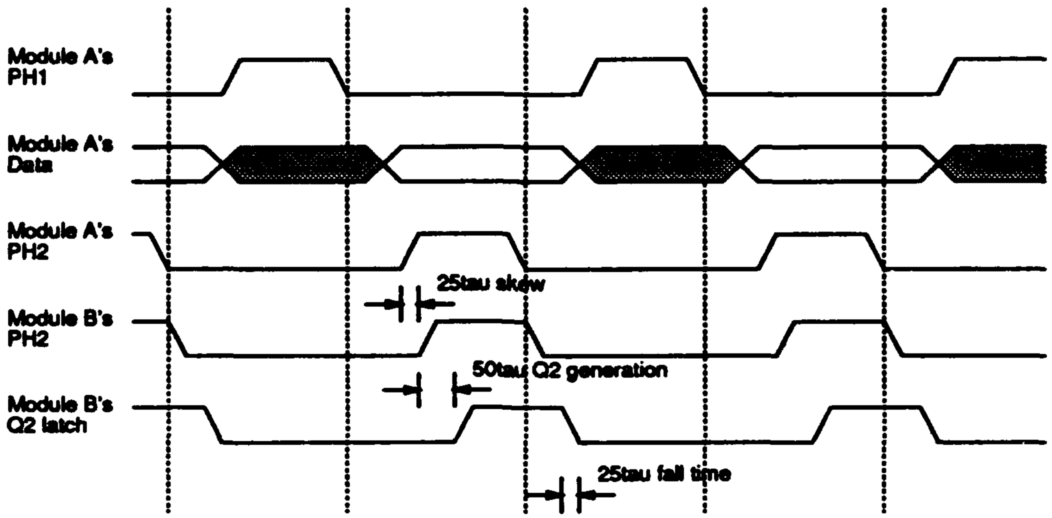


Figure 6.1: Where the nonoverlap time goes

V1	Valid during phase 1. S1 signals are the outputs of latches clocked by PH2 or *signals combinatorially derived from other* V1 signals.
V2	Valid during PH2.
Q1	Qualified by PH1. Q1 signals are asserted only during PH1. PH1 is a Q1 signal. Other Q1 signals are derived by ANDing PH1 with V1 signals.
Q2	Qualified by PH2.
S1	Setup during PH1. Occasionally, a signal originating at a PH2 latch will have a propagation time longer than 1/2 a clock period. Such signals will met the setup time for a PH1 latch, but cannot be used to generate a Q1 signal. Note that all V1 signals are S1 signals, so a V1 signal may safely be supplied to any input requiring an S1 signal.
S2	Setup during PH2.

The remaining restrictions are:

1. Maximum delay to a Q1 (Q2) signal from PH1 (PH2) is 50τ. Delay is measured from 50% of input to 50% of output.

2. Maximum rise(fall) time (10% to 90%) of a Q signal is 25τ.

3. There are no minimum delay constraints.

4. All combinatorial delays must be less than 250τ from PH1 or PH2 except where expressly noted. This is to allow a very simple timing verifier to operate.

5. Each latch must be supplied with a Q1 clock and S1 data or a Q2 clock and S2 data.

6. All feedback paths must be broken by a latch.

7. All signals sourced by a module should be V2.

8. All internal modules should latch their inputs on PH2, operate for zero or more cycles, and then drive their outputs starting in PH1.

9. Modules should assume that their inputs are S2, as they may still be changing during PH2, on account of clock skew between modules. They will be stable 100τ before the end of PH2.

10. All modules used should use only fully restored static CMOS logic.

11. All signals should be pulled to VDD or GND at all times. No precharged or pseudo-NMOS circuits should be employed. All transmission gates should include both N and P devices.

12. All modules should be designed so that all storage nodes are static during PH1. If the clock is stopped with PH1 high, no data should be lost.

13. All static latch feedback loops should connect only to the gates of transistors to avoid dynamic charge sharing.

6.2 Top Level Microarchitecture

With the design guidelines stated, the details of the microarchitecture can be presented, starting at the top level. The block diagram, as presented in chapter 4, did not provide any details of the wiring between modules. It is redrawn in figure 6.2 to explicitly call attention to the module interconnect. A representative of each modules is shown is figure 6.3, along with the intermodule signals used.

The chip interface consists of support signals, such as clock and reset, together with the signals used to communicate with other switching elements.

Name	I/O	Description
clock	I	The single phase clock.
reset	I	The chip reset.
d[4:0][7:0]	I	The port data in.
q[4:0][7:0]	O	The port data out.

6.3 Timing Generation

The timing generator tim_tim provides a chip wide timebase for flit relative timing. Since there are 8 phits per flit, the timing module provides 8 outputs, phitTime[7..0]. All are V2 signals. PhitTime[0] corresponds to the time when the first phit of a flit appears on the physical channel between IC's.

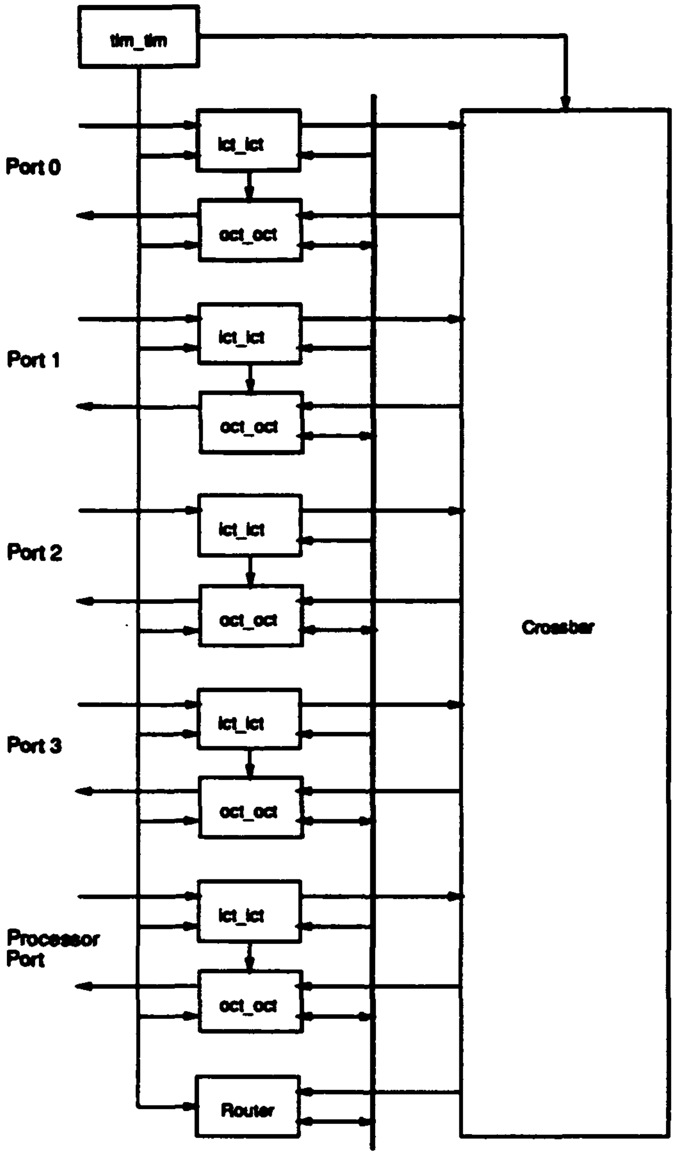


Figure 6.2: Top Level Module Block Interconnect

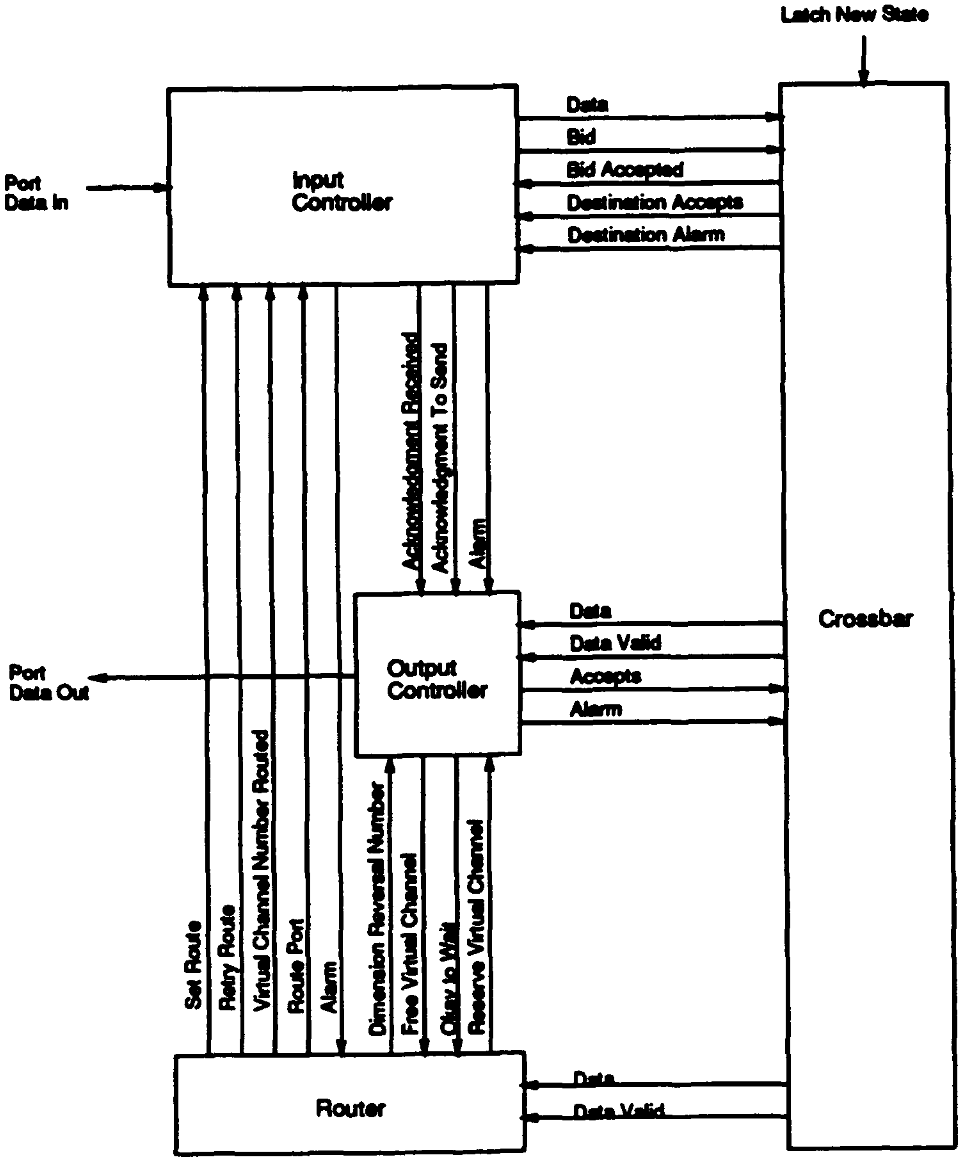


Figure 6.3: Top Level Inter-Module Signalling

6.4 Input controller - ict_ict

Module I/O

Name	I/O	Description
d[7..0]	I	One port's worth of physical input wires to the chip.
q[7..0]	O	Flit data sent to the crossbar
bid[2..0]	O	The output controller or router to send the flit to.
bid_accepted	I	The crossbar accepted the bid, i.e. a connection has made between the input controller and the desired output port.
dst_accepts	I	The connected output controller accepted the flit sent.
dst_alarm	I	The connected output controller is down.
alarm	O	The input controller has detected some form of error, possibly parity, CRC, alarm received, or other protocol violation.
alarms[4..0]	I	The alarms from all input controllers.
ack_recvd[2..0]	O	The last acknowledgment received.
ack_to_send[2..0]	O	The acknowledgment that the paired output controller should send.
set_route	I	A control input from the router, it is used to indicate that a route has been established.
retry_route	I	A control input from the router, it is used to indicate that a route has not been established and should be rerequested later.
vc_being_routed[2..0]	I	The virtual channel that the router is currently updating.
route_port[2..0]	I	When a route is established, this is the target exit port.

Description

The input controller is composed of only of submodules, no logic is implemented at this level. The block level interconnect is shown in figure 6.4. The submodules include:

input port, which captures incoming data,

checker, which checks incoming data,

low address generation, which provides a phit offset into the flit storage,

dual ported RAM, where the flits are stored,

address map, which converts a flit sequence number into a pointer,

pointer file, where the FIFO pointers for each virtual channel are kept,

load control, which controls the entry of flits into the FIFOs, and

unloader, which removes flits from the FIFOs and maintains channel state.

6.4.1 Input Port - ip_ip

Module I/O

Name	I/O	Description
d[7..0]	I	One port's worth of physical input wires to the chip.
q[7..0]	O	Flit data sent to the dual ported RAM
raw_q[7..0]	O	Flit data sent to the flit checker
ack[2..0]	O	The acknowledgment just received, it is sent to the paired output controller.

Description

The input port latches the external data using ph1. This "raw" data is sent to the checker for verification of parity, CRC, and alarm fields. The data is

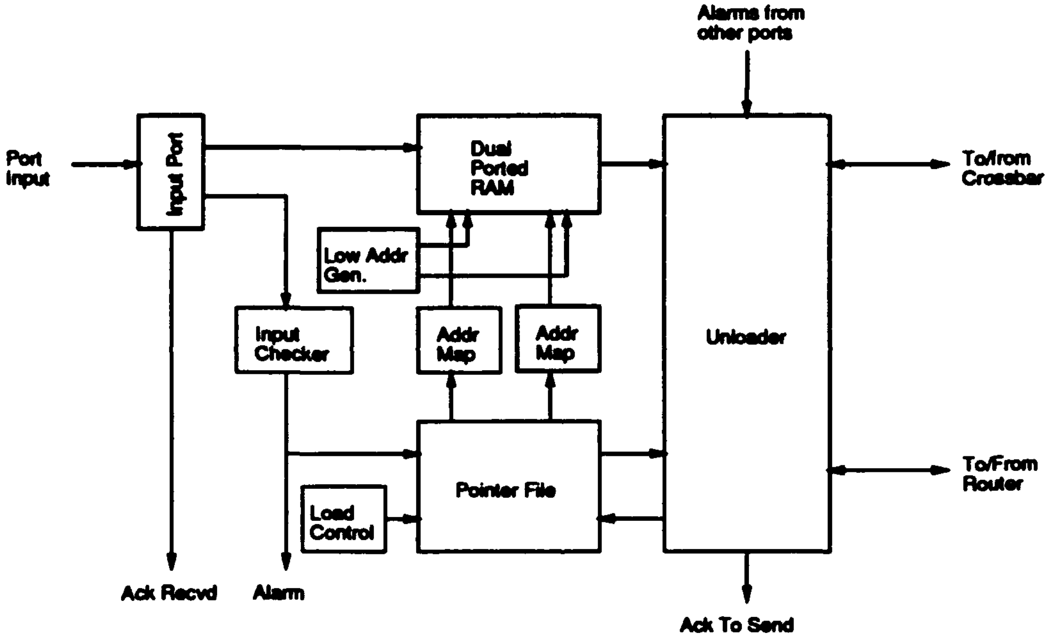


Figure 6.4: Input Controller Block Diagram

then delayed a couple of phits and sent to the dual ported RAM. The delay is necessary to allow the pointer file to retrieve the correct write pointer. It also allows the checker to suppress the increment of the write pointer, thus preventing errored flits from affecting the unloader. Note that an errored flit will be written into the dual ported RAM using a possibly errored VC, so an extra flit's worth of storage must be added to each input buffer to account for this overwriting.

6.4.2 Flit Checker - ict_check

Module I/O

Name	I/O	Description
d[7..0]	I	The raw phit stream from the input port.
alarm	O	Alarm is asserted when any error is detected.

Description

The checker checks the parity every clock period. It also monitors the alarm bit sent with each flit. If either condition ever becomes true, the alarm is set.

6.4.3 Pointer File - pf_pf

Module I/O

Name	I/O	Description
alarm	I	The alarm signal from this input controller's checker.
load_vc[2..0]	I	The virtual channel of the incoming flit
unload_vc[2..0]	I	The virtual channel for the outgoing flit
ack_vc[2..0]	I	The virtual channel of the flit just acknowledged
incr_ack	I	Control signal which causes the acknowledgment count for a virtual channel to increment.
load_opcode[2..0]	I	Operation desired by the loader.
unload_opcode[2..0]	I	Operation desired by the unloader.
read_ptr[5..0]	O	The sequence number of the flit to be read from the dual ported RAM.
write_pointer[5..0]	O	The sequence number of the flit to written to the dual ported RAM.
flits_to_send[4..0]	O	Indicates which virtual channels are non-empty.
send_ack[4..0]	O	Indicates which virtual channels have forwarded flits and not yet acknowledged them.

Description

The pointer file maintains 4 counters for each input virtual channel: a write pointer, an acknowledgment count, a head-of-packet read pointer, and a data read pointer. It also outputs two signals per virtual channel: flits_to_send and send_ack.

Two read pointers, not one, are needed to handle retransmissions for the tail ends of long packets. When a packet needs to be retransmitted, the head flit must again be sent to the router. After the head flit is transmitted, the effective read pointer might need change to point at any location in the input ring. The two read pointers are combined together to form an effective read pointer in the following way:

- if the head-of-packet read pointer is less than the number of head flits, the effective read pointer is the head-of-packet read pointer.
- if the head-of-packet pointer is equal to the number of head flits, the effective read pointer is the sum of the head-of-packet pointer and the data read pointer.

When the read pointer is to be incremented, the head-of-packet pointer is incremented until it reaches the number of head flits, then the data read pointer is incremented.

The write pointer is used by the loader to place flits into the dual ported RAM. When the write pointer is greater than the effective read pointer, the signal flits_to_send is asserted.

The acknowledgment counter is not externally visible. When the effective read pointer is greater than the acknowledgment count, the signal send_ack is asserted.

Backing up

This is a bit tricky. Suppose that the data portion of the input buffer is n flits long and the write pointer is w. We know that one flit of the n may be inadvertently overwritten when input channel errors occur. We also know that some number of flits may sent prior to an acknowledgment being sent. Call this threshold number t. So, the $t+1$ flits after the current write pointer may be overwritten. We can therefore only back the read pointer to be $w-(n-(t+1)) = w-n+t+1$ flits.

6.4.4 Pointer File Load Controller - pf_ctl

Module I/O

Name	I/O	Description
opcode[2..0]	O	The opcode to the pointer file for flit loading.

Description

The pointer file load controller initiates a fetch of the write pointer for each incoming flit and increments the write pointer.

6.4.5 Dual Ported Ram - dpr_dpr

Module I/O

Name	I/O	Description
d[7..0]	I	Data in
q[7..0]	O	Data out
ra[9..0]	I	Read address
wa[9..0]	I	Write address
we	I	Write enable

Description

This is a simple dual ported RAM.

6.4.6 Address Generator - ict_addr

Module I/O

Name	I/O	Description
ra[2..0]	O	Lower three bits of read address.
wa[2..0]	O	Lower three bits of write address.

Description

The address generator provides the phit-within-a-flit address portion of the dual ported RAM's read and write addresses.

6.4.7 Pointer to Address Mapper - ict_pfmap

Module I/O

Name	I/O	Description
pf_addr[5..0]	I	Address from the pointer file.
dpr_addr[3..0]	O	Address to the dual ported RAM.

Description

The pointer to address mapper converts the 0 to 63 range of pointers to a number between 0 and 15 for the dual ported RAM. The function is 0 maps to 0, all other numbers n map to (n mod 15) + 1.

6.4.8 Unload Controller - ict_unload

Module I/O

Name	I/O	Description
flits_to_send[2..0]	I	These inputs from the pointer file indicate that a virtual channel queue is non-empty and that a flit should be forwarded.
send_ack[2..0]	I	These inputs from the pointer file indicate that an acknowledgment for a virtual channel should be sent when one of its flits is forwarded.
make_replica	O	Forces the flit being forwarded to be of type tokenReplica.
gen_replica	O	When the tail of a packet is lost, no token will be in the input queue and one must be generated. This signal forces one to be generated.
pf_vc[2..0]	O	The virtual channel sent to the pointer file to obtain its current read pointer, increment it, back it up, or clear it.
dpr_vc[2..0]	O	Current virtual channel number for the dual ported RAM. This is simply a delayed version of pf_vc.
dpr_q[7..0]	I	The output of the dual ported RAM which the unloader needs to determine the flit's kind.
opcode[2..0]	O	Opcode to the pointer file.
bid[2..0]	O	Desired exit port out of the crossbar,
bid_accepted	I	The crossbar made the crosspoint.
dst_accepts	I	The flit was accepted by the output controller.
dst_alarm	I	The alarm condition at the output controller.
src_alarm	I	The alarm condition in the input loader.

Name	I/O	Description
alarms[4..0]	I	The alarms from all output controllers, used to suppress acknowledgments as late as possible.
route_port[2..0]	I	The exit port to use for a virtual channel.
vc_being_routed[2..0]	I	The virtual channel the routing is updating.
set_route	I	The router established the route.
retry_route	I	The router was unable to establish the route, but the packet could wait. Try again later.
ack_to_send[2..0]	O	The acknowledgment to send out through the paired output controller.
incr_ack	O	Increment the acknowledgment count in the pointer file.

Description

The unload controller manages most of the state associated with an incoming virtual channel. The state information is:

vc_state Either *idle*, *waitingForRoute*, or *active*.

vc_backed_up A boolean which indicates if this virtual channel had had to be backed up and rerouted.

vc_to_port The target exit port.

Initially, the state for all virtual channels is idle, not backed up.

When the loader places a flit into the input buffer for a virtual channel, the flits_to_send signal for that virtual channel becomes asserted, Each flit time, the unloader scans the virtual channels, looking for a virtual channel with flits to send that is either in the idle or active states. If the virtual channel is in the idle state, the flit is sent to the router and the channel state changed to waiting_for_route. If the virtual channel was in the active state, the flit is sent to the destination port found in the vc_to_port table.

For flits of type head and data, the unloader sends the flit across the crossbar by first presenting a bid to the crossbar. If the bid is accepted and the destination accepts the flit, the read pointer is incremented in the pointer file. This has the side effect of updating the flits_to_send signal.

For flits of type token, the situation is more complex. The output controller will refuse the flit as long as there are any outstanding unacknowledged flit transmissions for this virtual channel. This is to ensure that all the flits in the packet have been copied to two places before freeing up the input channel. Thus, when the input controller sends a flit of type token across the crossbar, the state associated with the virtual channel is initialized.

For all kinds of flits, if the destination responds with an alarm, the read pointer in the pointer file is backed up, the channel state is changed to idle, and vc_backed_up set. This eventually causes the packet or packet fragment to be re-routed. When the token for the packet is finally transmitted, the assertion of vc_backed_up causes the type of the token to be coerced to replica.

The unloader only notices the failure of the physical input channel when a virtual channel is in the active state and there are no more flits to send. The unloader forces the generation of a token of type replica and then initializes the state of the virtual channel.

Lastly, the unloader generates acknowledgments. When a flit is successfully sent to an output controller, the virtual channel number and output controller number are placed in a 3 deep pipeline which is shifted every flit time. At the end of the pipeline, the output controller is checked to see if an alarm has been set. If there is no alarm and the pointer file indicates that an acknowledgment should be sent, it is sent and the acknowledgment count in the pointer file incremented.

Acknowledgments for flits of type token are handled slightly differently, as the state of the virtual channel is initialized as soon as the flit made it to the output controller. The fact that it was of type token is also placed in the 3 deep pipeline. At the end of the pipeline, an acknowledgment is always sent when the type token indicator appears.

6.4.9 Token Generation - ict_token

Module I/O

Name	I/O	Description
d[7..0]	I	The flit data in.
make_replica	I	If the flit is of type tokenUnique, coerce it to be a replica.
gen_replica	I	Create a new flit of type tokenReplica.
dpr_vc[2..0]	I	The virtual channel number to use when generating a token.
q[7..0]	O	The flit data out.

Description

This module handles the coercion and the generation of flits to type tokenReplica.

6.5 Crossbar

Module I/O

Name	I/O	Description
d[4..0][7..0]	I	The data entering the crossbar
q[5..0][7..0]	O	The data leaving the crossbar
bid[4..0][2..0]	I	The bids from the input controllers
column_valid[5..0]	O	Indicates to the output controller that the data appearing is valid.
bid_accepted[4..0]	O	Indicates to the input controller that a crosspoint was established.
dst_accepts_in[5..0]	I	The signal from each output controller which indicates that the output controller accepted the flit.
dst_accepts_out[4..0]	O	This is the dst_accepts_in signal brought back over the crosspoint.
alarm_in[5..0]	I	The destination's alarm.
alarm_out[4..0]	O	The destination's alarm brought back through the crosspoint to the input controller.

Description

This module is composed of an array of crosspoints (xb_elt) and a token injector (xb_inject). Nearly the entire crossbar is simply the two dimensional array of crossbar elements, so all of the crosspoint allocation logic is distributed into each crosspoint. This makes the simulation model a bit hard to understand, but allows a simpler physical realization.

The crossbar is organized into rows and columns. Inputs to the crossbar are sent along the rows, outputs are taken from the columns.

6.5.1 Token injector - xb_inject

Module I/O

Name	I/O	Description
latch_new_state	I	A once per flit signal used to change the state of the crossbar.
tokens[4..0]	O	Only one of these will be asserted. The asserted token is used to determine which input controller has the highest priority bid.

Description

In order to ensure fair access to the crossbar, an input controller is selected at random to have the highest priority bid.

6.5.2 Crossbar crosspoints - xbar_elt

Module I/O

Name	I/O	Description
d[7..0]	I	The row data to be moved across the crosspoint.
q[7..0]	O	The data driven onto the column.
bid[2..0]	I	The bid from the input controller.
inject	I	The initial token from the injector, giving this row highest priority.
token_in	I	Priority daisy-chain in.
token_out	O	Priority daisy-chain out.
column_available	I	Column was not allocated by the router. Not used.
latch_new_state	I	Places the crosspoint in high impedance and stores the new crosspoint setting.
column_valid	O	A crosspoint will provide valid data to the output controller or router attached to this column.
bid_accepted	O	A crosspoint in this row accepted the bid.
dst_accepts_in	I	The flit was accepted by the output controller, in.
dst_accepts_out	O	The flit was accepted by the output controller, out.
alarm_in	I	The alarm for the output controller attached to this row.
alarm_out	O	The alarm sent to the input controller from an established crosspoint.

Description

An input controller broadcasts a bid along a row of crosspoints. Each crosspoint knows its column number. The crosspoint compares the bid against its column number. If there is a match, that crosspoint is responsible for sending status information back to the input controller.

An allocation token is injected along one row of the crossbar each flit time. When a crosspoint has the allocation token, if it matched the input controller's bid, it establishes the crosspoint and asserts column_valid. If the crosspoint did not match, it forwards the allocation token in its column.

If the allocation token should return to the same row as it was injected into, that crosspoint deasserts column_valid.

6.6 Output controller - oct_oct

6.6.1 Module I/O

Name	I/O	Description
d[7..0]	I	Data from the crossbar.
column_valid	I	An indication from the crossbar that crosspoint to an input controller has been established.
send_alarm	I	The alarm indicator from the paired input controller.
ack_to_send[2..0]	I	From the paired input controller, this indicates which input virtual channel has successfully replicated a flit further into the network and should now be acknowledged.
ack_recvd[2..0]	I	From the paired input controller, this signal indicates the availability of one more flit's worth of storage in the queue across the physical channel.
pad	O	The data to be sent to the output pads.
flit_valid	O	The output controller asserts this signal when it accepts the flit.
reserve_vc[2..0]	I	From the router, this reserves a virtual channel at the given DR level.
dr_from_rt[5..0]	I	The DR number of the packet currently being routed.
free_vc	O	Asserted when the output controller has free virtual channels.
okay_to_wait	O	Asserted when the the DR number of the packet being routed is less than or equal to the DR number of all the packets assigned to virtual channels.

6.6.2 Description

The output controller is made of two submodules: the output filter and the output packager.

6.6.3 Output filter - oct_filter

Module I/O

Name	I/O	Description
d[7..0]	I	Data from the crossbar.
column_valid	I	An indication from the crossbar that crosspoint to an input controller has been established.
send_alarm	I	The alarm indicator from the paired input controller.
ack[2..0]	I	From the paired input controller, this signal indicates the availability of one more flit's worth of storage in the queue across the physical channel. Corresponds the *ack_recvd signal in the* surrounding module.
q[7..0]	O	Data to the output packager.
flit_valid	O	The output controller asserts this signal when it accepts the flit.
reserve_vc	I	From the router, this reserves a virtual channel at the given DR level.
dr[5..0]	I	The DR number of the packet currently being routed.
free_vc	O	Asserted when the output controller has free virtual channels.
okay_to_wait	O	Asserted when the the DR number of the packet being routed is less than or equal to the DR number of all the packets assigned to virtual channels.

Description

The output filter performs three functions. It implements flow control. It maps <input port, input vc> tuples to output vc's. Finally, it provides output virtual channel information to the router.

Flow control is implemented by keeping a count of the number of flits which have been sent on an output virtual channel which have not yet been acknowledged. When the count exceeds some threshold, the output filter will stop accepting flits from the crossbar for that channel. The ack input will cause the count for that virtual channel to decrement.

The mapping function is straightforward. For each output virtual channel, its associated input port and input virtual channel are remembered in a table. When a flit arrives, the table is searched for a match and the vc field in the flit updated.

The output filter remembers the DR number of each packet associated with an active output virtual channel. When the router is determining where to send a packet, it first broadcasts the packet's DR number to all the output filters. Each filter responds to the router with an indication of free virtual channels, If there are none, the output filter must tell the router if the router is able to wait for one based on the DR number presented and the DR numbers stored.

When the router determines that it wants a packet to go out a given output port, it reserves a virtual channel in the output filter. This is to allow the output filter to remember the DR number for the channel in time for the next routing decision. If the output filter waited for the head flit to arrive before establishing the channel, the state could wind up being inconsistent.

When the head flit of a packet arrives, the output filter looks for a reserved virtual channel with a matching DR number and completes the map.

The output filter will not accept a flit of type token if the number of unacknowledged flits is non-zero. Once the flit is accepted, the output virtual channel is marked as *reclaimNextAck*. The next acknowledgment for that virtual channel will cause the channel to be freed.

6.6.4 Output packager - oct_package

Module I/O

Name	I/O	Description
d[7..0]	I	Data from the output filter.
send_alarm	I	The alarm indicator from the paired input controller.
ack[2..0]	I	From the paired input controller, this indicates which input virtual channel has successfully replicated a flit further into the network and should now be acknowledged. This corresponds to ack_to_send in the surrounding module.
pad[7..0]	O	The data to be sent to the output pads.

Description

The output packager splices the acknowledgment and alarm fields into the outgoing flit. It then recomputes the CRC field for the flit. Finally, it computes parity for the first and last phits of the flit.

6.7 Router

6.7.1 Module I/O

Name	I/O	Description
our_x[5..0]	I	The x coordinate of the switching element.
our_y[5..0]	I	The y coordinate of the switching element.
our_z[5..0]	I	The z coordinate of the switching element.
dr_to_oct[5..0]	O	The DR number of the packet currently being routed.
okay_to_wait[4..0]	I	From each output controller, these are asserted when the the DR number of the packet being routed is less than or equal to the DR number of all the packets assigned to virtual channels.
free_vcs_from_oct[4..0]	I	From each output controller, these are asserted when the output controller has free virtual channels.
reserve_vc[4..0]	O	This reserves a virtual channel in a particular output controller.
q[7..0]	O	Data input to the crossbar. Currently not used.
bid[2..0]	O	Output port select to the crossbar. Currently not used.
bid_accepted	I	Not used.
dst_accepts	I	Not used.
d[7..0]	I	The data lines from the crossbar carrying the head flit of the packet to be routed.
column_valid	I	An indication from the crossbar that a crosspoint from an input controller has been established.
dst_accepts_out	O	Currently always asserted, it indicates that router will attempt to route this packet.
dst_alarm	I	Always deasserted and currently ignored.

Name	I/O	Description
alarm_stub	O	Always deasserted, as the router cannot have an alarm.
route_port[2..0]	O	The exit port fro the packet being routed.
vc_being_routed[2..0]	O	The input virtual channel of the packet being routed.
set_route[4..0]	O	Causes the input controller to record the routing information and begin forwarding the packet.
retry_route[4..0]	O	Causes the input controller to retry the routing the packet later.
alarm[4..0]	I	The alarm conditions of all the ports.

6.7.2 Description

When an input controller decides that a packet needs to be routed, it send the head flit to the router. The router parses the flit into destination X,Y,Z coordinates and DR number. It extracts the input port number and input virtual channel number. It then attempts to find an output port.

To find the output port, it first computes the desired exit port, using dimension ordered routing. If the desired port is either non-functional or if exiting the desired port would result in a backtrack, a new port is chosen at random. If the exit port has no free virtual channels and the packet is unable to wait owing to dimensional reversal constraints, a new port is again chosen.

Finally, if the desired exit port has a free virtual channel, the router reserves the channel and communicates the exit port to the input controller. If the exit port had no free channels, the router tells the input controller to retry the route request.

6.8 Summary

In this chapter, we stated the goals of the architecture, namely to show that the unique token protocol was realizable using standard cell technology. Design guidelines in support of the goals were developed, describing clocking, intermodule communication, and circuit-design style. The microarchitecture was then presented at the top level and successively refined. This microarchitecture has been implemented as a register-transfer model using the M modeling language. The verification of this model will be discussed in the next chapter.

Chapter 7

Design Verification

We have developed a new protocol, the unique token protocol, for reliably moving data through a network. To show that the protocol is realizable, a switching-element microarchitecture was designed and presented in the previous chapter. This microarchitecture has been implemented as a register-transfer level (RTL) model using the M modeling language[1]. In this chapter, we describe the validation of this model and the subsequent observations.

The goal of the verification is to ensure that the model functions as a router for a two dimensional mesh network and that the implementation delivers packets according to the unique token protocol, under both fault-free and faulted conditions.

7.1 Verification Procedure

The verification process was done in four stages and, as bugs were found and corrected, repeated several times until the model was verified. The first stage is a single chip test. The original purpose of this test was to aid in bringing up a single device, along with the various traffic sources and sinks. Once the design stabilized, this test evolved into a confidence test of the switching element by injecting random traffic for an extended period of time. This stage of testing does not not involve any fault recovery.

[1] Mentor Graphics, Silicon Design Division

The second stage of testing connects two devices together. The purpose is to expose the discrepancies between the artificial traffic generated by the test fixture and the actual traffic between two switching elements.

In the third stage of testing, a 2 × 2 mesh is constructed and traffic generators attached to all spare ports. This provided a slightly longer path for messages to travel and served as a base case for the fourth stage.

The final stage of testing is again a 2 × 2 mesh. Traffic is sourced only from a single processor port and is sent to the processor port on the opposite corner of the mesh. Since the router first attempts dimension ordered routing, this traffic would travel along a preferred route. The first link along this route is sabotaged and the protocol allowed to adapt to the fault and recover. The timing of the fault is changed from run to run.

7.2 Test Jigs and Scaffolding

Support code falls into three classes: port stubs, fixtures, and post-mortem sanity checks.

A port stub provides a source of traffic to send into a port on a chip. It is capable of injecting a variable number of packets into all the virtual channels within a port. The length of the packet is chosen at random between some easily changed lower and upper bounds. The packet consists of a head flit, a data flit containing a unique identifier, some number of data flits containing random data, and a token flit.

The destination address contained in each packet may be bounded by min/max X,Y,Z pairs independently in each port stub. The port stub will not generate traffic to its local processor port. It also receives and acknowledges flits to allow the flow control algorithm to work. Without such support, the chip model hangs after it has sent a few flits.

All flits sent and received are written in human-readable form into a log file, to make debugging easier. The format is also readily parsed by computer programs for more automated verification.

Three major and several minor variants of the test fixture have been written. The first fixture is a single chip with port stubs attached at all five locations. This fixture was heavily used throughout the course of writing and debugging the model. The second fixture has two chip models interconnected with port stubs at all other open locations.

The third fixture implements a 2 × 2 mesh. This fixture was used mainly to verify the reliability protocol. The stubs are attached to the processor ports of the chips. Two variants have been constructed, one with stubs at the edges of the mesh, the other with the edge ports grounded. This fixture is capable of injecting a fault along one the links. The fault time may be varied at run time via a parameter on the simulator command line. This makes it possible to write a shell script to make repeated simulator runs, stepping through fault times.

A program, *sane*, has been written to verify the correct operation of the switching element. It first compiles a list of packets which have been injected, then compiles the list of packets extracted. It discards runt packets, those that consist only of a head flit and a token. It splices together packets from the packet fragments arriving tagged as type replica. Finally, it compares the packets sent against the packets received for length and contents. If all packets sent have a corresponding packet received and vice-versa, then the test has been passed.

7.3 Testing Results

Three different configurations were tested under normal operation. A single chip was injected with 720 packets, two chips with 9 packets, and the 2 × 2 mesh with 640 packets. All configurations passed.

A 2 × 2 mesh was used as the basis for the fourth test. The source of traffic was the processor port on the lower left. It sent six messages of random lengths to the processor at the upper right. Since the router is primarily dimension ordered, the usual path includes the switching element on the lower right. At some programmable time, the wires leading from the node on the lower left to the node on the lower right were forced to all zeros.

This basic test was run from within a shell script which stepped the failure time from time zero until a time after all six packets had been received. After each pass, the sanity checker was run and the results written into a file. All runs passed the sanity checker.

7.4 Summary

The register transfer model of the reliable switching element has been simulated in a variety of configurations and stresses. It has shown correct functionality under these conditions. From this, we conclude that the model constructed is a faithful implementation of the unique token protocol.

Chapter 8

Conclusions

This thesis presented a new protocol, the unique token protocol, for reliable communication using replication in a network. It contrasted the scalability of this protocol with existing end-to-end protocols. It described the organization and microarchitecture of a switching element implementing the protocol. Finally, it presented the results of the design verification simulations. This work has led to several conclusions concerning the protocol and possible implementations.

The unique token protocol can be used in a wide variety of network topologies. It consumes less network bandwidth than end-to-end protocols when acknowledgments are taken into account. It offers excellent scalability, with storage requirements dominated by $O(n)$ terms, where n is the number of processors.

What the protocol does under some boundary conditions is not well defined in this thesis. For example, the recovery procedure to use when the destination fails and the message is undeliverable is not given, as it has strong interactions with the method of adaptive routing chosen. One has to decide that the destination is truly unreachable and not that the bulk of the physical channels leading into the final switching element are down. This is currently handled by delivering the packet to the nearest processor when the packet's dimension reversal number overflows. Software will then be used to probe for alternate routes or return the packet if none exist. A future research topic is to provide semantics for undeliverable packets.

The implementation provided several insights. The key one is that the protocol is indeed implementable without a dramatic increase in logic complexity over an adaptive, deadlock-free switching element. Indeed, the changes are confined mainly to the input controller.

The second insight is that containment of network errors can lead to increased latency. The switching element has state which is altered by incoming packets. The model, as constructed, did not allow any critical state to be altered until the incoming data could be certified error-free. Thus, the progress of the data is slowed by the error checking process. This increased latency may be dealt with by providing wider data paths, which collapses the time from start of flit until verification. Secondly, one could checkpoint the critical state and backtrack upon error detection. The amount of critical state is heavily determined by the network topology and the complexity of mechanisms required for adaptive routing and deadlock avoidance. Some topologies, such as indirect connection networks, are intrinsically deadlock free and may have less complex checkpointing circuits.

A second cause of latency in the model is the use of a centralized router. By centralizing the router, several input controllers contend for its use. If the typical packet size is large, this contention will be minimized. The centralization was done for practical reasons in order to avoid the construction of some form of distributed router. The problem is that the router not only decides where to send the packet, it must also allocate resources in the form of channel bandwidth. Distributing the resource allocation decision is possible through the introduction of a second crossbar array. This second crossbar would allow an input controller to lock the output controller and allocate a virtual channel without interference from other input controllers.

Appendix A

Buffer Storage Requirements of End-to-End Protocols

End-to-end protocols require storage in the source. We need to understand how that storage requirement changes with machine size. Consider a node in the middle of a very long chain of $n+1$ nodes as shown in figure A.1. Each physical channel has bandwidth ω, where ω has units of msgs/sec. The system has an aggregate bandwidth of $n\omega$, measured in channel-msgs/sec. When this aggregate bandwidth is evenly distributed over the nodes, each node may use ω channel-msgs/sec.

Let A be a probability distribution function (PDF) describing the likelihood that a node wishes to send a message to another node, passing through d channels. ($d = 0$ implies no message sent.)

$$\forall d, d \geq 0, \; A(d) = a_d \tag{A.1}$$

Since it is a PDF, we know that:

$$\sum_{d=0}^{\infty} a_d = 1 \tag{A.2}$$

Each node would be using system bandwidth equal to:

$$\sum_{d=0}^{\infty} \omega d a_d \tag{A.3}$$

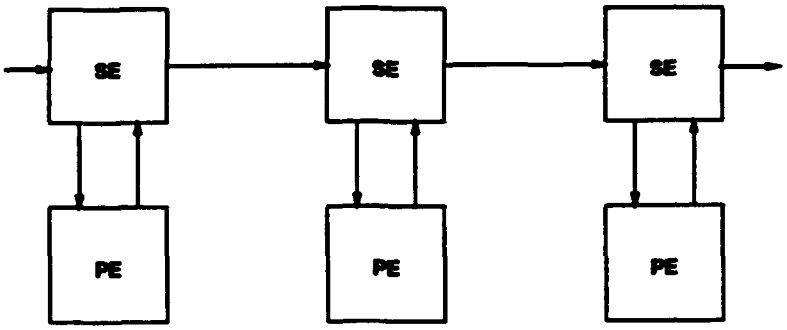


Figure A.1: Simple Traffic Model

which must be equal to ω under uniform maximum loads. We can then write:

$$\sum_{d=0}^{\infty} d a_d = 1 \tag{A.4}$$

Let the internode delay on a channel be ϵ. Let γ be the time it takes a node to receive a message and generate an acknowledgment. The total round trip for a message and its acknowledgment to a node d channels away is:

$$2d\epsilon + \gamma \tag{A.5}$$

The expected amount of storage required to hold copies of messages sent to a node d channels away is simply the probability of sending a message d channels away × channel bandwidth × duration:

$$a_d \omega (2d\epsilon + \gamma) \tag{A.6}$$

The total expected storage is then:

$$\sum_{d=0}^{\infty} a_d \omega (2d\epsilon + \gamma) = 2\omega\epsilon \sum_{d=0}^{\infty} d a_d + \omega\gamma \sum_{d=0}^{\infty} a_d \tag{A.7}$$

Using equations A.2 and A.4, this reduces to:

$$\omega(2\epsilon + \gamma) \tag{A.8}$$

The expected amount of storage used to buffer messages is constant, irrespective of the actual PDFs used to model the traffic sources. Note that we have only calculated the expected amount of storage used under uniform conditions. When a node can consume a disproportionately large share of the system bandwidth, its expected storage requirements increase. This can be seen mathematically by introducing a load factor into equation A.4:

$$\sum_{d=0}^{\infty} da_d = \phi \tag{A.9}$$

which results in expected storage of:

$$\omega(2\epsilon\phi + \gamma) \tag{A.10}$$

Intuitively, we can bound ϕ by considering a network where only one node is sending traffic, and that is to a destination node over the longest possible path. For 2D mesh networks, these nodes are on opposite corners of the machine. ϕ is then simply the number of hops needed to get from source to destination. For such a mesh network, we have:

$$\phi = 2(\sqrt{n} - 1) \tag{A.11}$$

Substituting A.11 into A.10 yields an expected storage requirement of:

$$\omega(4\epsilon(\sqrt{n} - 1) + \gamma) \tag{A.12}$$

We see that the expected storage requirements in 2D meshes for end-to-end protocols grow as $O(\sqrt{n})$.

Bibliography

[1] Athas, W.C., and Seitz, C.L., "Multicomputers: Message-Passing Concurrent Computers," IEEE Computer, Vol 21, No 8, August 1988, pp. 9-24.

[2] Bartlett, K.A., Scantkebury, R.A., and Wilkinson, P.T., "A note on reliable Full-Duplex Transmission Over Half-Duplex Links", CACM, 12, No. 5, 260, 1969

[3] BBN Advanced Computers, Inc., Butterfly Parallel Processor Overview, BBN Report No 6148, March 1986.

[4] Chen, M.S. and Shin, K.G., "Adaptive Fault-Tolerant Routing in Hypercube Multicomputers", IEEE Transactions on Computers, Vol. 39, No. 12, December 1990, pp. 1406-1416.

[5] Chen, M.S. and Shin, K.G., "Message Routing in an Injured Hypercube", Proc. Third Conf. Hypercube Concurrent Computer Appl., January 1988, pp. 312-317.

[6] Dally, W.J. "Wire-Efficient VLSI Multiprocessor Communication Networks," Proceedings of the Stanford Conference on Advanced Research in VLSI, Paul Losleben, ed., MIT press, March 1987, pp. 391-415.

[7] Dally, W.J., "Network and Processor Architecture for Message-Driven Computing," in VLSI and Parallel Processing, R. Suaya and G. Birtwistle eds., Morgan Kaufmann, to appear 1989.

[8] Dally, W.J., "Express Cubes: Improving the Perfomance of k-ary n-cube Interconnection Networks", IEEE Transactions on Computers, to appear 1991.

[9] Dally, W.J. and Seitz, C.L., "Deadlock Free Message Routing in Multiprocessor Interconnection Networks", IEEE Transactions on Computers, Vol C-36, No. 5, May 1987, pp. 547-553.

[10] Frey, A.H. and Fox, G.C., "Problems and Approaches for a Teraflop Processor", Caltech Report C^3P-606.

[11] Gordon, J.M. and Stout, Q.F., "Hypercube Message Routing in the Presence of Faults", Proc. Third Conf. Hypercube Concurrent Computer Appl., January 1988, pp. 318-327.

[12] Kermani, P., and Kleinrock, L., "Virtual Cut-Through: A New Computer Communication Switching Technique." Computer Networks, Vol 3, 1979, pp. 267-286.

[13] Karol, M.J., Hluchy, M.G., and Morgan, S.P., "Input Versus Output Queueing on a Space-Division Packet Switch." IEEE Transactions on Communications, Vol COM-35, No 12, December 1987, pp. 1347-1356.

[14] Lee, C.T. and Hayes, J.P., "Routing and Broadcasting in Faulty Hypercube Computers", Proc. Third Conf. Hypercube Concurrent Computer Appl., January 1988, pp. 346-354.

[15] Liu and Leyland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, Vol 20, No 1, January 1973, pp. 46-61.

[16] Davies, D.W., Barber, D.L.A., Price, W.L., and Solomonides, C.M., *Computer Networks and Their Protocols*, John Wiley and Sons Ltd, 1979.

[17] Mailhot, J.N., A Comparative Study of Routing and Flow-Control Strategies in k-ary n-cube Networks, Massachusetts Institute of Technology, SB Thesis, May 1988.

[18] Tamir, Y., and Frazier, G.L., "High-Performance Multi-Queue Buffers for VLSI Communication Switches," 15th annual ACM/IEEE Symposium on Computer Architecture, June 1988, pp. 343-354.

[19] Tanenbaum, A.S. *Computer Networks, Second Edition*, Englewood Cliffs, NJ:Prentice-Hall, 1988.